(12) UK Patent Application (19) GB (11) 2 339 040 (13) A

(43) Date of A Publication 12.01.2000

(21) Application No 9907221.7

(22) Date of Filing 29.03.1999

(30) Priority Data
(31) 09053127 (32) 31.03.1998 (33) US

(51) INT CL[7]
G06F 9/302

(52) UK CL (Edition R)
G4A AVL

(56) Documents Cited
WO 97/22924 A1 WO 97/22923 A1 WO 97/22921 A1

(58) Field of Search
UK CL (Edition Q) G4A AVL
INT CL[6] G06F 9/302

(71) Applicant(s)
Intel Corporation
(Incorporated in USA - Delaware)
2200 Mission College Boulevard, Santa Clara, California 95052, United States of America

(72) Inventor(s)
Patrice Roussel
Ticky Thakkar

(74) Agent and/or Address for Service
Langner Parry
High Holborn House, 52-54 High Holborn, LONDON, WC1V 6RR, United Kingdom

(54) Abstract Title
Executing partial-width packed data instructions

(57) In a data processing system arranged for executing scalar packed data instructions, a processor includes a plurality of registers, a register renaming unit coupled to the plurality of registers, a decoder coupled to the register renaming unit, and a partial-width execution unit coupled to the decoder. The register renaming unit provides an architectural register file to store packed data operands each of which include a plurality of data elements. The decoder is configured to decode a first and second set of instructions that each specify one or more registers in the architectural register file. Each of the instructions in the first set of instructions specify operations to be performed on all of the data elements stored in the one or more specified registers. In contrast, each of the instructions in the second set of instructions specify operations to be performed on only a subset of the data element stored in the one or more specified registers. The partial-width execution unit is configured to execute operations specified by either of the first or the second set of instructions.

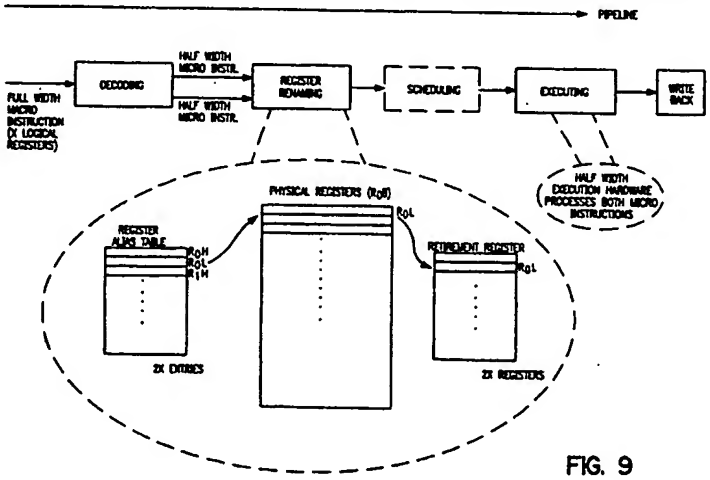


FIG. 9

At least one drawing originally filed was informal and the print reproduced here is taken from a later filed formal copy.

GB 2 339 040 A

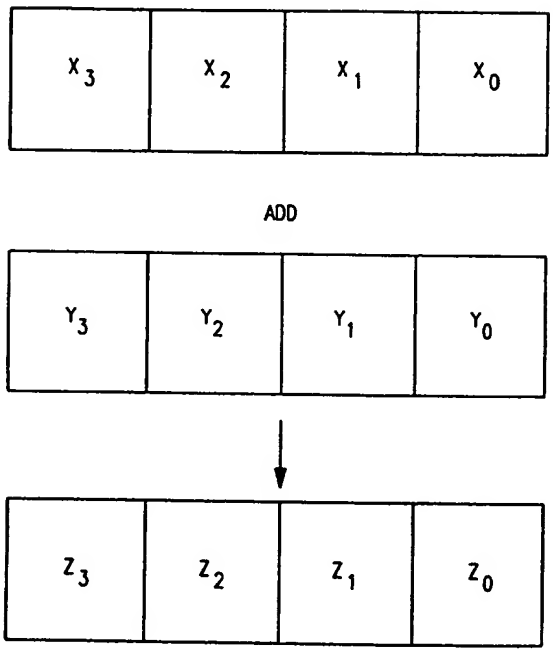


FIG. 1
PRIOR ART

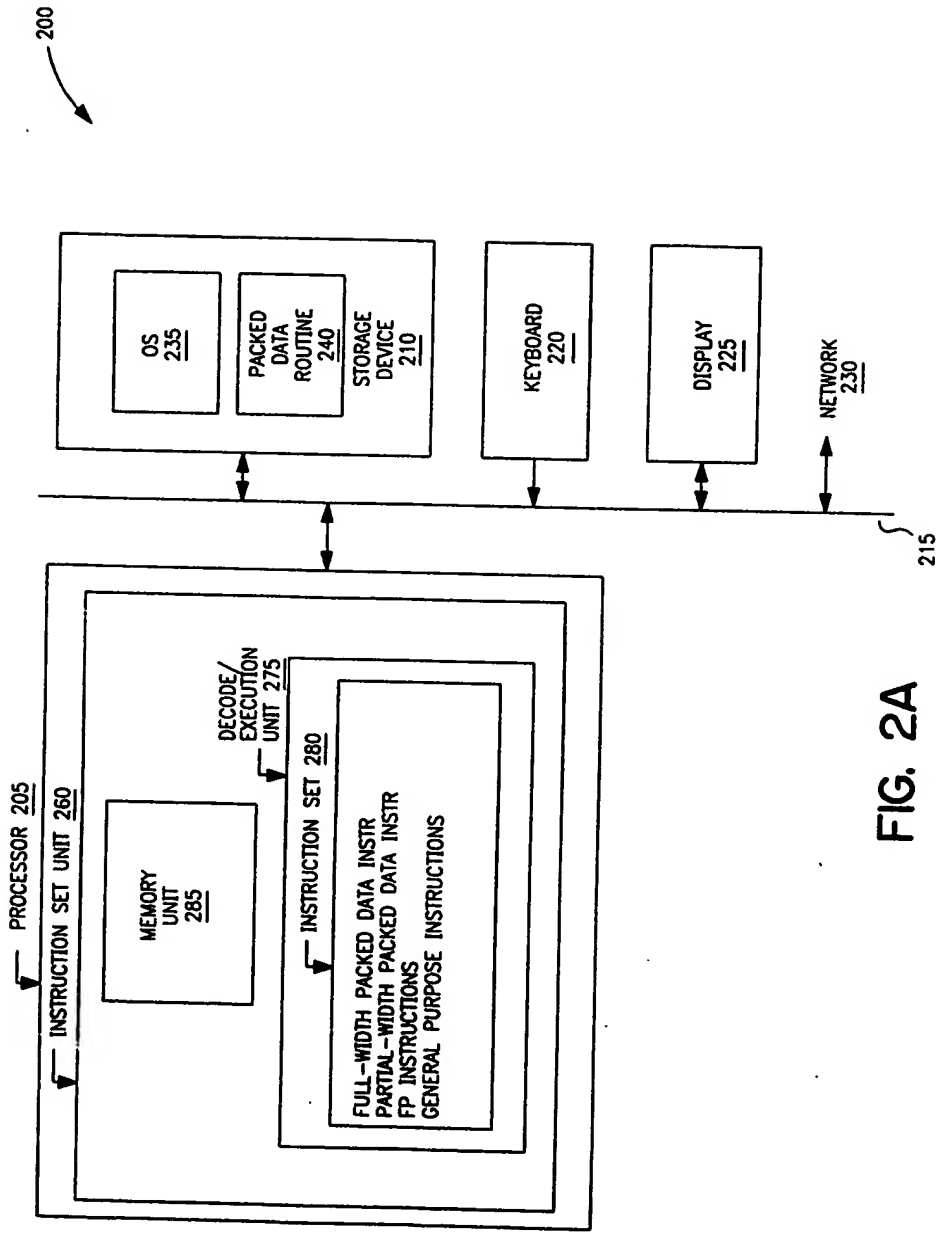

200
OS 235
PACKED DATA ROUTINE 240
STORAGE DEVICE 210
KEYBOARD 220
DISPLAY 225
NETWORK 230
215
PROCESSOR 205
INSTRUCTION SET UNIT 260
MEMORY UNIT 285
DECODE/ EXECUTION UNIT 275
INSTRUCTION SET 280
FULL-WIDTH PACKED DATA INSTR
PARTIAL-WIDTH PACKED DATA INSTR
FP INSTRUCTIONS
GENERAL PURPOSE INSTRUCTIONS

FIG. 2A

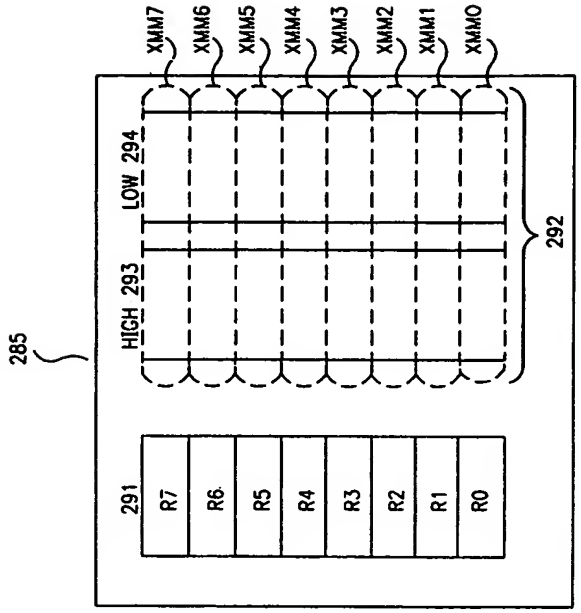

285
291
R7
R6
R5
R4
R3
R2
R1
R0
HIGH 293
LOW 294
292
XMM7
XMM6
XMM5
XMM4
XMM3
XMM2
XMM1
XMM0

FIG. 2C

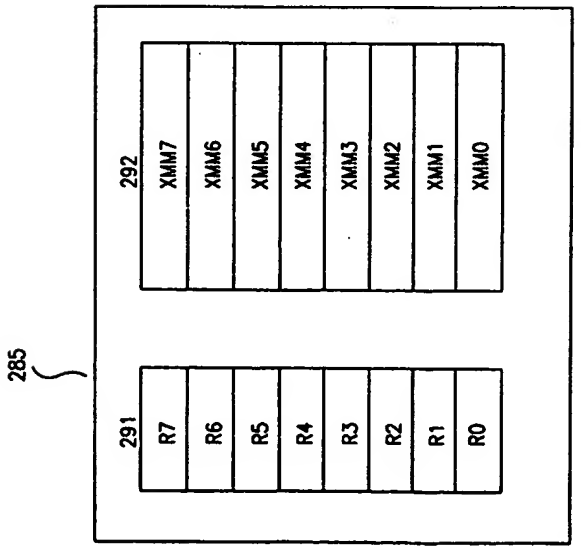

285
291
R7
R6
R5
R4
R3
R2
R1
R0
292
XMM7
XMM6
XMM5
XMM4
XMM3
XMM2
XMM1
XMM0

FIG. 2B

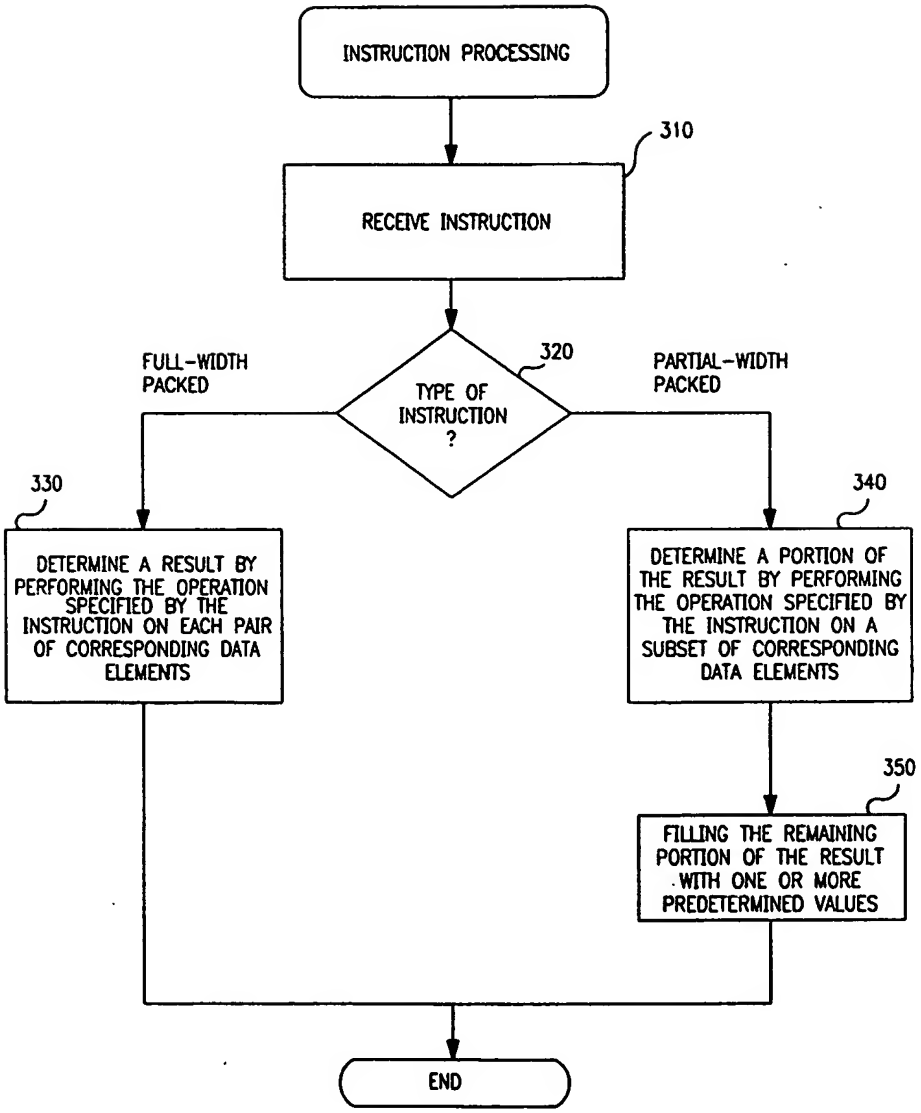

INSTRUCTION PROCESSING
310
RECEIVE INSTRUCTION
320
TYPE OF INSTRUCTION ?
FULL-WIDTH PACKED
PARTIAL-WIDTH PACKED
330
DETERMINE A RESULT BY PERFORMING THE OPERATION SPECIFIED BY THE INSTRUCTION ON EACH PAIR OF CORRESPONDING DATA ELEMENTS
340
DETERMINE A PORTION OF THE RESULT BY PERFORMING THE OPERATION SPECIFIED BY THE INSTRUCTION ON A SUBSET OF CORRESPONDING DATA ELEMENTS
350
FILLING THE REMAINING PORTION OF THE RESULT WITH ONE OR MORE PREDETERMINED VALUES
END

FIG. 3

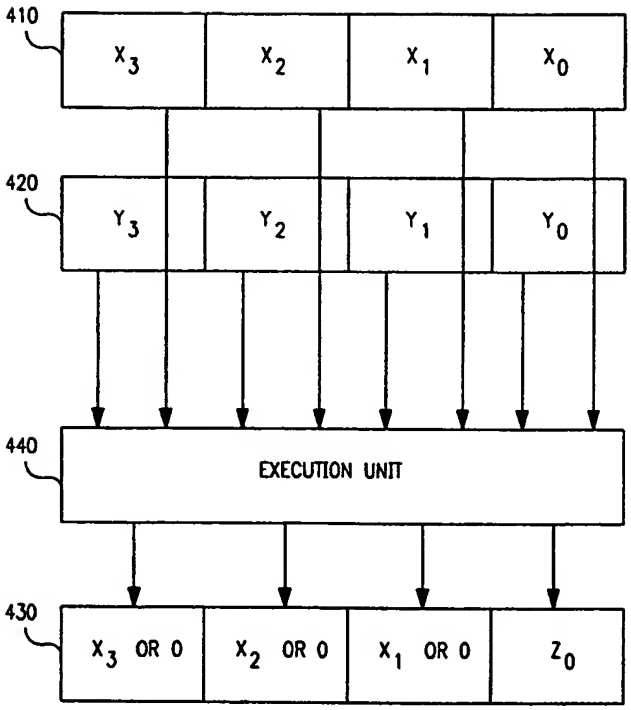

410
X3
X2
X1
X0
420
Y3
Y2
Y1
Y0
440
EXECUTION UNIT
430
X3 OR 0
X2 OR 0
X1 OR 0
Z0

FIG. 4

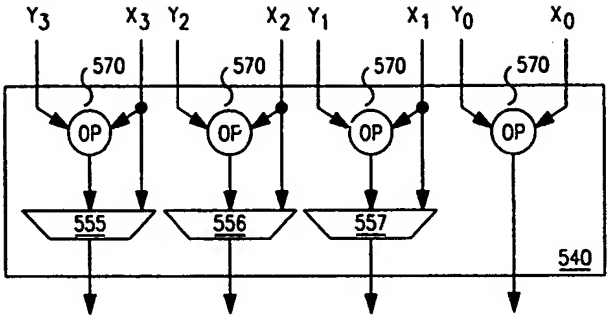


FIG. 5A

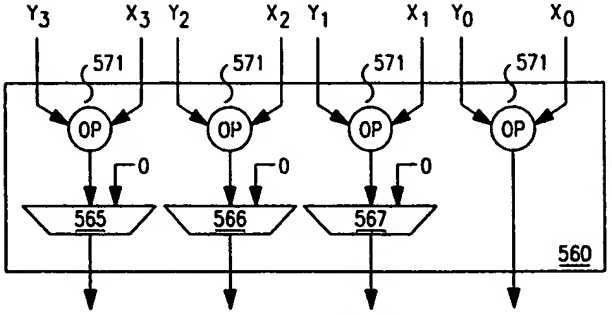


FIG. 5B

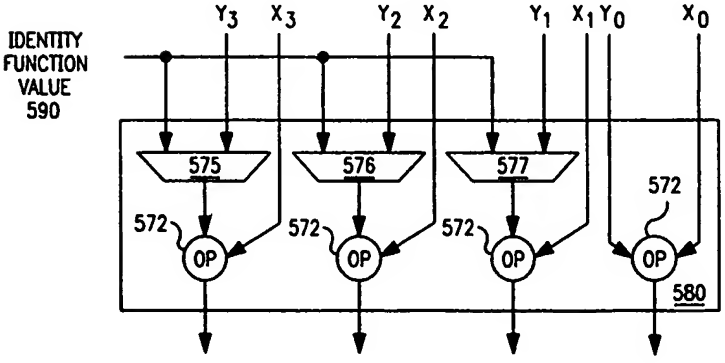


FIG. 5C

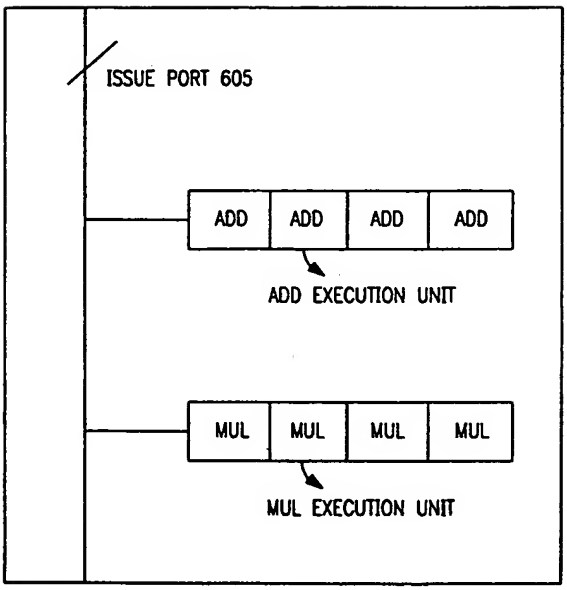

ISSUE PORT 605
ADD
ADD
ADD
ADD
ADD EXECUTION UNIT
MUL
MUL
MUL
MUL
MUL EXECUTION UNIT

FIG. 6

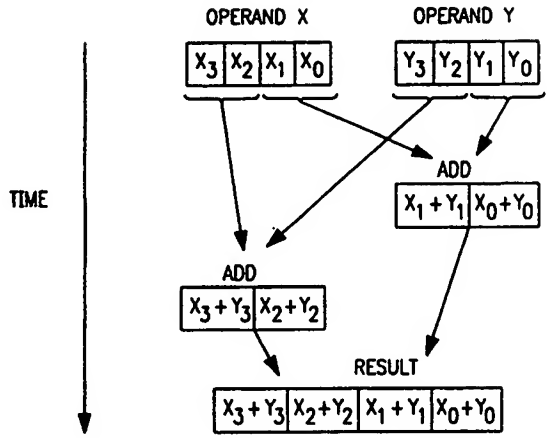
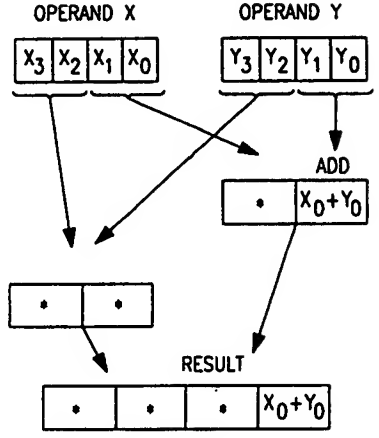


FIG. 7A



• = Corresponding data element in X or Y, NaN, Ø, or other predetermined value

FIG. 7B

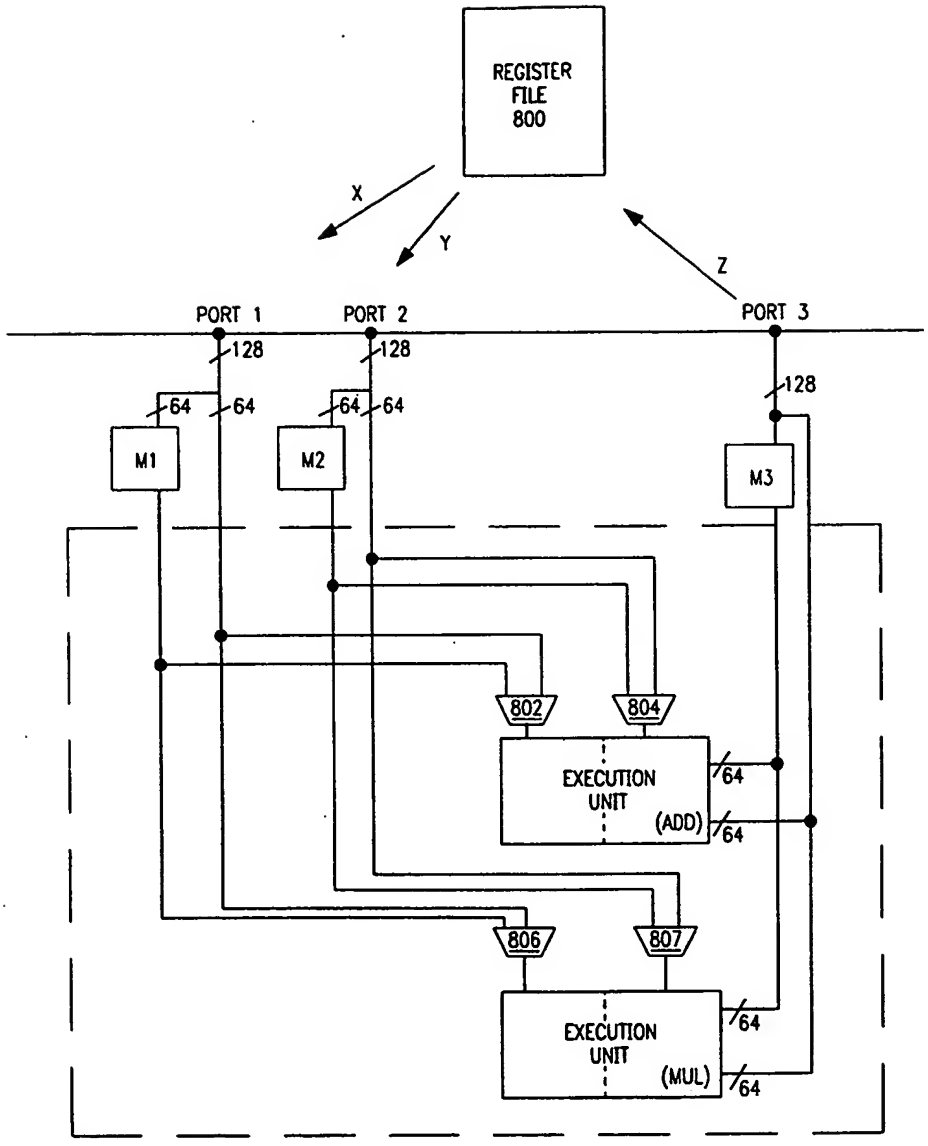

REGISTER FILE 800
X
Y
Z
PORT 1
PORT 2
PORT 3
128
128
128
64
64
64
64
M1
M2
M3
802
804
EXECUTION UNIT
(ADD)
64
64
806
807
EXECUTION UNIT
(MUL)
64
64

FIG. 8A

TIME	128-bit instruction	Performed on 64-bit data
T	ADD X, Y	ADD X_0 Y_0 ADD X_1 Y_1
T+1		ADD X_2 Y_2 ADD X_3 Y_3
T+1	MUL X, Y	MUL X_0 Y_0 MUL X_1 Y_1
T+2		MUL X_2 Y_2 MUL X_3 Y_3

FIG. 8B

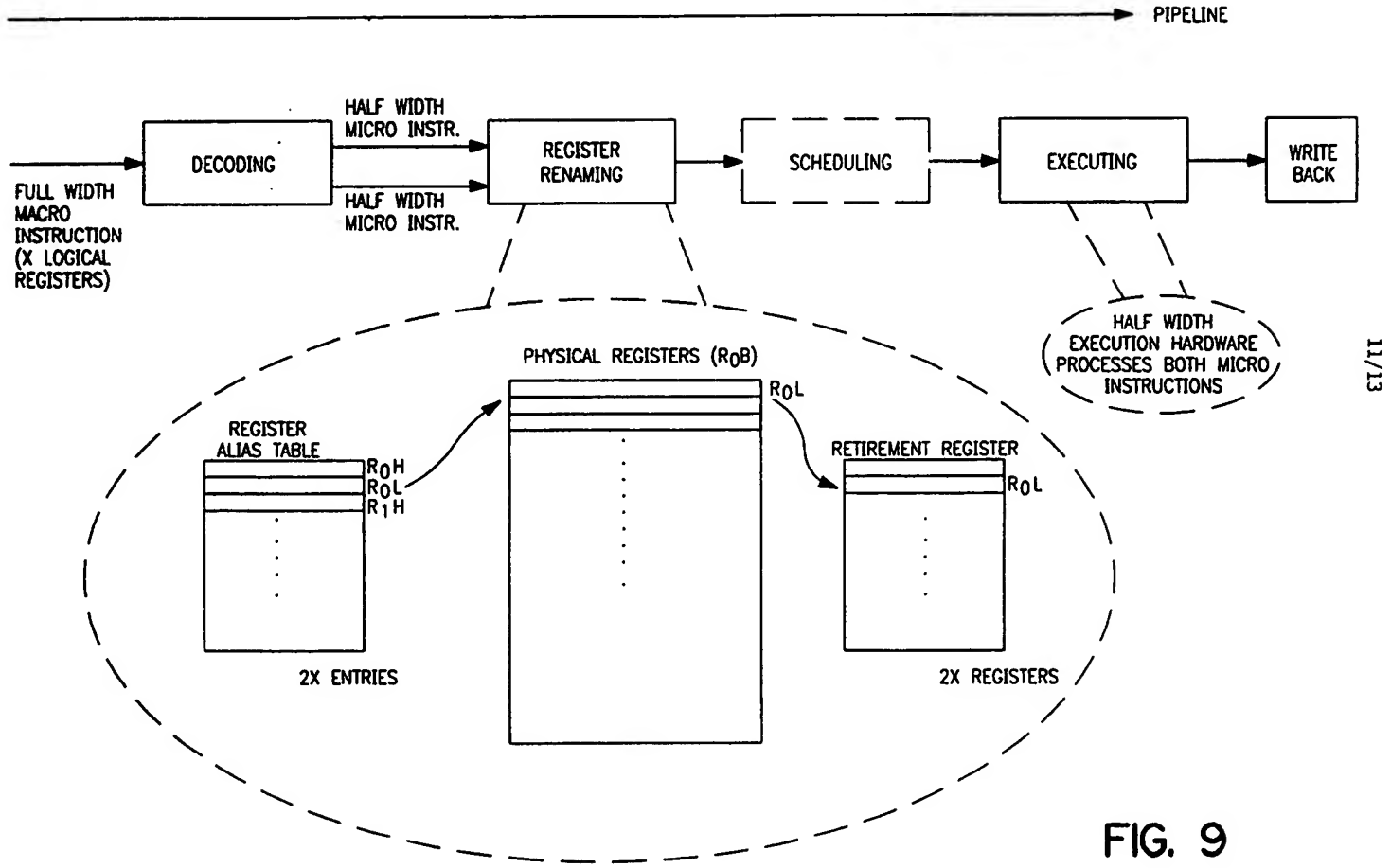

PIPELINE
FULL WIDTH MACRO INSTRUCTION (X LOGICAL REGISTERS)
DECODING
HALF WIDTH MICRO INSTR.
HALF WIDTH MICRO INSTR.
REGISTER RENAMING
SCHEDULING
EXECUTING
WRITE BACK
HALF WIDTH EXECUTION HARDWARE PROCESSES BOTH MICRO INSTRUCTIONS
PHYSICAL REGISTERS (R0B)
R0L
REGISTER ALIAS TABLE
R0H
R0L
R1H
RETIREMENT REGISTER
R0L
2X ENTRIES
2X REGISTERS

FIG. 9

TIME	128-BIT INSTRUCTION	64-BIT INSTRUCTION
T	ADD X,Y	ADD X_L,Y_L
T+N		ADD X_H,Y_H

FIG. 10

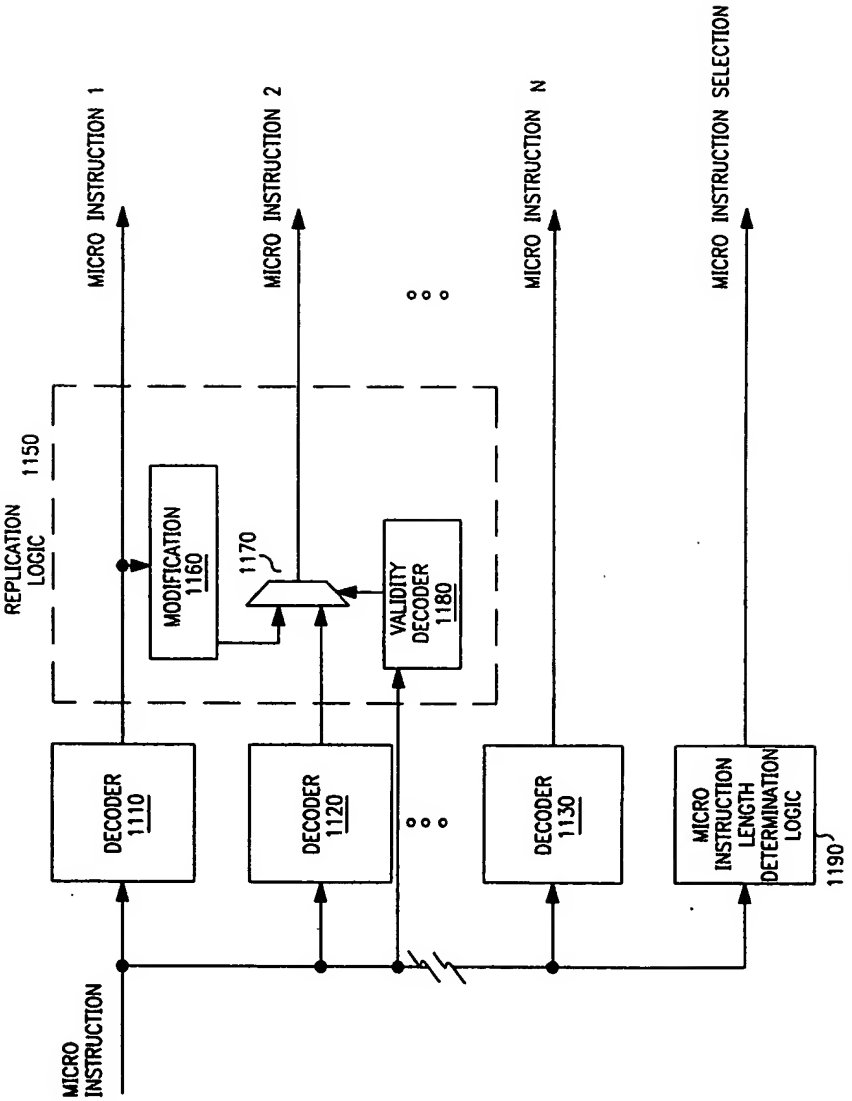

MICRO INSTRUCTION 1
MICRO INSTRUCTION 2
MICRO INSTRUCTION N
MICRO INSTRUCTION SELECTION
REPLICATION LOGIC 1150
MODIFICATION 1160
1170
VALIDITY DECODER 1180
DECODER 1110
DECODER 1120
DECODER 1130
MICRO INSTRUCTION LENGTH DETERMINATION LOGIC 1190
MICRO INSTRUCTION

FIG. 11

EXECUTING PARTIAL-WIDTH PACKED DATA INSTRUCTIONS

FIELD OF THE INVENTION

The invention relates generally to the field of computer systems. More particularly, the invention relates to a method and apparatus for efficiently executing partial-width packed data instructions, such as scalar packed data instructions, by a processor that makes use of SIMD technology, for example.

BACKGROUND OF THE INVENTION

Multimedia applications such as 2D/3D graphics, image processing, video compression/decompression, voice recognition algorithms and audio manipulation, often require the same operation to be performed on a large number of data items (referred to as "data parallelism"). Each type of multimedia application typically implements one or more algorithms requiring a number of floating point or integer operations, such as ADD or MULTIPLY (hereafter MUL). By providing macro instructions whose execution causes a processor to perform the same operation on multiple data items in parallel, Single Instruction Multiple Data (SIMD) technology, such as that employed by the Pentium® processor architecture and the MMx™ instruction set, has enabled a significant improvement in multimedia application performance (Pentium® and MMx™ are registered trademarks or trademarks of Intel Corporation of Santa Clara, CA).

SIMD technology is especially suited to systems that provide packed data formats. A packed data format is one in which the bits in a register are logically divided into a number of fixed-sized data elements, each of which represents a separate value. For example, a 64-bit register may be broken into four 16-bit elements, each of which represents a separate 16-bit value. Packed data instructions may then separately manipulate each element in these packed data types in parallel.

Referring to Figure 1, an exemplary packed data instruction is illustrated. In this example, a packed ADD instruction (e.g., a SIMD ADD) adds corresponding data elements of a first packed data operand, X, and a second packed data operand, Y, to produce a packed data result, Z, i.e., $X_0+Y_0=Z_0$, $X_1+Y_1=Z_1$, $X_2+Y_2=Z_2$, and $X_3+Y_3=Z_3$. Packing many data elements within one register or memory location and employing parallel hardware execution allows SIMD architectures to perform multiple operations at a time, resulting in significant performance improvement. For instance, in this example, four individual results may be obtained in the time previously required to obtain a single result.

While the advantages achieved by SIMD architectures are evident, there remain situations in which it is desirable to return individual results for only a subset of the packed data elements.

SUMMARY OF THE INVENTION

A method and apparatus are described for executing partial-width packed data instructions. According to one aspect of the invention, a processor includes a plurality of registers, a register renaming unit coupled to the plurality of registers, a decoder coupled to the register renaming unit, and a partial-width execution unit coupled to the decoder. The register renaming unit provides an architectural register file to store packed data operands each of which include a plurality of data elements. The decoder is configured to decode a first and second set of instructions that each specify one or more registers in the architectural register file. Each of the instructions in the first set of instructions specify operations to be performed on all of the data elements stored in the one or more specified registers. In contrast, each of the instructions in the second set of instructions specify operations to be performed on only a subset of the data element stored in the one or more specified registers. The partial-width execution unit is configured to execute operations specified by either of the first or the second set of instructions.

Other features and advantages of the invention will be apparent from the accompanying drawings and from the detailed description.

BRIEF DESCRIPTION OF THE DRAWINGS

The invention is described by way of example and not by way of limitation with reference to the figures of the accompanying drawings in which like reference numerals refer to similar elements and in which:

Figure 1 illustrates a packed ADD instruction adding together corresponding data elements from a first packed data operand and a second packed data operand.

Figure 2A is a simplified block diagram illustrating an exemplary computer system according to one embodiment of the invention.

Figure 2B is a simplified block diagram illustrating exemplary sets of logical registers according to one embodiment of the invention.

Figure 2C is a simplified block diagram illustrating exemplary sets of logical registers according to another embodiment of the invention.

Figure 3 is a flow diagram illustrating instruction execution according to one embodiment of the invention.

Figure 4 conceptually illustrates the result of executing a partial-width packed data instruction according to various embodiments of the invention.

Figure 5A conceptually illustrates circuitry for executing full-width packed data instructions and partial-width packed data instructions according to one embodiment of the invention.

Figure 5B conceptually illustrates circuitry for executing full-width packed data and partial-width packed data instructions according to another embodiment of the invention.

Figure 5C conceptually illustrates circuitry for executing full-width packed data and partial-width packed data instructions according to yet another embodiment of the invention.

Figure 6 illustrates an ADD execution unit and a MUL execution unit capable of operating as four separate ADD execution units and four separate MUL execution units, respectively, according to an exemplary processor implementation of SIMD.

Figures 7A-7B conceptually illustrate a full-width packed data operation and a partial-width packed data operation being performed in a "staggered" manner, respectively.

Figure 8A conceptually illustrates circuitry within a processor that accesses full width operands from logical registers while performing operations on half of the width of the operands at a time.

Figure 8B is a timing chart that further illustrates the circuitry of Figure 8A.

Figure 9 conceptually illustrates one embodiment of an out-of-order pipeline to perform operations on operands in a "staggered" manner by converting a macro instruction into a plurality of micro instructions that each processes a portion of the full width of the operands.

Figure 10 is a timing chart that further illustrates the embodiment described in Figure 9.

Figure 11 is a block diagram illustrating decoding logic that may be employed to accomplish the decoding processing according to one embodiment of the invention.

DETAILED DESCRIPTION

A method and apparatus are described for performing partial-width packed data instructions. Herein the term "full-width packed data instruction" is meant to refer to a packed data instruction (e.g., a SIMD instruction) that operates upon all of the data elements of one or more packed data operands. In contrast, the term "partial-width packed data instruction" is meant to broadly refer to a packed data instruction that is designed to operate upon only a subset of the data elements of one or more packed data operands and return a packed data result (to a packed data register file, for example). For instance, a scalar SIMD instruction may require only a result of an operation between the least significant pair of packed data operands. In this example, the remaining data elements of the packed data result are disregarded as they are of no consequence to the scalar SIMD instruction (e.g., the remaining data elements are don't cares). According to the various embodiments of the invention, execution units may be configured in such a way to efficiently accommodate both full-width packed data instructions (e.g., SIMD instructions) and a set of partial-width packed data instructions (e.g., scalar SIMD instructions).

In the following detailed description, for purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the invention. It will be apparent, however, to one of ordinary skill in the art that these specific details need not be used to practice

the invention. In other instances, well-known devices, structures, interfaces, and processes have not been shown or are shown in block diagram form.

Justification of Partial-Width Packed Data Instructions

Considering the amount of software that has been written for scalar architectures (e.g., single instruction single data (SISD) architectures) employing scalar operations on single precision floating point data, double precision floating point data, and integer data, it is desirable to provide developers with the option of porting their software to architectures that support packed data instructions, such as SIMD architectures, without having to rewrite their software and/or learn new instructions. By providing partial-width packed data instructions, a simple translation can transform old scalar code into scalar packed data code. For example, it would be very easy for a compiler to produce scalar SIMD instructions from scalar code. Then, as developers recognize portions of their software that can be optimized using SIMD instructions, they may gradually take advantage of the packed data instructions. Of course, computer systems employing SIMD technology are likely to also remain backwards compatible by supporting SISD instructions as well. However, the many recent architectural improvements and other factors discussed herein make it advantageous for developers to transition to and exploit SIMD technology, even if only scalar SIMD instructions are employed at first.

Another justification for providing partial-width packed data instructions is the many benefits which may be achieved by operating on only a subset of a full-width operand, including reduced power consumption, increased speed, a clean exception model, and increased storage. As illustrated below, based on an indication provided with the partial-width packed data instruction, power savings may be achieved by selectively shutting down those of the hardware units that are unnecessary for performing the current operation.

Another situation in which it is undesirable to force a packed data instruction to return individual results for each pair of data elements includes arithmetic operations in an environment providing partial-width hardware. Due to cost and/or die limitations, it is common not to provide full support for certain arithmetic operations, such as divide. the By its nature, the divide operation is very long, even when full-width hardware (e.g., a one-to-one correspondence between execution units and data elements) is implemented,. Therefore, in an environment that supports only full-width packed data operations while providing partial-width hardware, the latency becomes even longer. As will be illustrated further below, a partial-width packed data operation, such as a partial-width packed data divide operation, may selectively allow certain portions of its operands to bypass the divide hardware. In this manner, no performance penalty is incurred by operating upon only a subset of the data elements in the packed data operands.

Additionally, exceptions raised in connection with extraneous data elements may cause confusion to the developer and/or incompatibility between SISD and SIMD machines. Therefore, it is advantageous to report exceptions for only those data elements upon which the instruction is meant to operate. Partial-width packed data instruction support allows a predictable exception model to be achieved by limiting the triggering of exceptional conditions to those raised in connection with the data elements being operated upon. or in which exceptions produced by extraneous data elements would be likely to cause confusion or incompatibility between SISD and SIMD machines.

Finally, in embodiments where portions of destination packed data operand is not corrupted as a result of performing a partial-width packed data operation, partial-width packed data instructions effectively provide extra register space for storing data. For instance, if the lower portion of the packed data operand is being operated upon, data may be stored in the upper portion and vice versa.

An Exemplary Computer System

Figure 2A is a simplified block diagram illustrating an exemplary computer system according to one embodiment of the invention. In the embodiment depicted, computer system 200 includes a processor 205, a storage device 210, and a bus 215. The processor 205 is coupled to the storage device 210 by the bus 215. In addition, a number of user input/output devices, such

as a keyboard 220 and a display 225 are also coupled to bus 215. The computer system 200 may also be coupled to a network 230 via bus 215. The processor 205 represents a central processing unit of any type of architecture, such as a CISC, RISC, VLIW, or hybrid architecture. In addition, the processor 205 may be implemented on one or more chips. The storage device 210 represents one or more mechanisms for storing data. For example, the storage device 210 may include read only memory (ROM), random access memory (RAM), magnetic disk storage mediums, optical storage mediums, flash memory devices, and/or other machine-readable mediums. The bus 215 represents one or more buses (e.g., AGP, PCI, ISA, X-Bus, EISA, VESA, etc.) and bridges (also termed as bus controllers). While this embodiment is described in relation to a single processor computer system, it is appreciated that the invention may be implemented in a multi-processor computer system. In addition while the present embodiment is described in relation to a 32-bit and a 64-bit computer system, the invention is not limited to such computer systems.

Figure 2A additionally illustrates that the processor 205 includes an instruction set unit 260. Of course, processor 205 contains additional circuitry; however, such additional circuitry is not necessary to understanding the invention. An any rate, the instruction set unit 260 includes the hardware and/or firmware to decode and execute one or more instruction sets. In the embodiment depicted, the instruction set unit 260 includes a decode/execution unit 275. The decode unit decodes instructions received by

processor 205 into one or more micro instructions. The execution unit performs appropriate operations in response to the micro instructions received from the decode unit. The decode unit may be implemented using a number of different mechanisms (e.g., a look-up table, a hardware implementation, a PLA, etc.).

In the present example, the decode/execution unit 275 is shown containing an instruction set 280 that includes both full-width packed data instructions and partial-width packed data instructions. These packed data instructions, when executed, may cause the processor 205 to perform full-/partial-width packed floating point operations and/or full-/partial-width packed integer operations. In addition to the packed data instructions, the instruction set 280 may include other instructions found in existing micro processors. By way of example, in one embodiment the processor 205 supports an instruction set which is compatible with Intel 32-bit architecture (IA-32) and/or Intel 64-bit architecture (IA-64).

A memory unit 285 is also included in the instruction set unit 260. The memory unit 285 may include one or more sets of architectural registers (also referred to as logical registers) utilized by the processor 205 for storing information including floating point data and packed floating point data. Additionally, other logical registers may be included for storing integer data, packed integer data, and various control data, such as a top of stack indication and the like. The terms architectural register and logical register are used herein to refer to the concept of the manner in which instructions specify a

storage area that contains a single operand. Thus, a logical register may be implemented in hardware using any number of well known techniques, including a dedicated physical register, one or more dynamically allocated physical registers using a register renaming mechanism (described in further detail below), etc.. In any event, a logical register represents the smallest unit of storage addressable by a packed data instruction.

In the embodiment depicted, the storage device 210 has stored therein an operating system 235 and a packed data routine 240 for execution by the computer system 200. The packed data routine 240 is a sequence of instructions that may include one or more packed data instructions, such as scalar SIMD instructions or SIMD instructions. As discussed further below, there are situations, including speed, power consumption and exception handling, where it is desirable to perform an operation on (or return individual results for) only a subset of data elements in a packed data operand or a pair of packed data operands. Therefore, it is advantageous for processor 205 to be able to differentiate between full-width packed data instructions and partial-width packed data instructions and to execute them accordingly.

Figure 2B is a simplified block diagram illustrating exemplary sets of logical registers according to one embodiment of the invention. In this example, the memory unit 285 includes a plurality of scalar floating point registers 291 (a scalar register file) and a plurality of packed floating point registers 292 (a packed data register file). The scalar floating point registers 291 (e.g., registers R0-R7) may be implemented as a stack referenced register file

when floating point instructions are executed so as to be compatible with existing software written for the Intel Architecture. In alternative embodiments, however, the registers 291 may be treated as a flat register file. In the embodiment depicted, each of the packed floating point registers (e.g., XMM0-XMM7) are implemented as a single 128-bit logical register. It is appreciated, however, wider or narrower registers may be employed to conform to an implementation that uses more or less data elements or larger or smaller data elements. Additionally, more or less packed floating point registers 292 may be provided. Similar to the scalar floating point registers 291, the packed floating point registers 292 may be implemented as either a stack referenced register file or a flat register file when packed floating point instructions are executed.

Figure 2C is a simplified block diagram illustrating exemplary sets of logical registers according to another embodiment of the invention. In this example, the memory unit 285, again, includes a plurality of scalar floating point registers 291 (a scalar register file) and a plurality of packed floating point registers 292 (a packed data register file). However, in the embodiment depicted, each of the packed floating point registers (e.g., XMM0-XMM7) are implemented as a corresponding pairs of high 293 and low registers 294. As will be discussed further below, it is advantageous for purposes of instruction decoding to organize the logical register address space for the packed floating point registers 292 such that the high and low register pairs differ by a single bit. For example, the high and low portions of XMM0-XMM7 may be

differentiated by the MSB. Preferably, each of the packed floating point registers 291 are wide enough to accommodate four 32-bit single precision floating point data elements. As above, however, wider or narrower registers may be employed to conform to an implementation that uses more or less data elements or larger or smaller data elements. Additionally, while the logical packed floating point registers 292 in this example each comprise corresponding pairs of 64-bit registers, in alternative embodiments each packed floating point register may comprise any number of registers.

Instruction Execution Overview

Having described an exemplary computer system in which one embodiment of the invention may be implemented, instruction execution will now be described.

Figure 3 is a flow diagram illustrating instruction execution according to one embodiment of the invention. At step 310, an instruction is received by the processor 205. At step 320, based on the type of instruction, partial-width packed data instruction (e.g., scalar SIMD instruction) or full-width packed data instruction (e.g., SIMD instruction), processing continues with step 330 or step 340. Typically, in the decode unit the type of instruction is determined based on information contained within the instruction. For example, information may be included in a prefix or suffix that is appended to an opcode or provided via an immediate value to indicate whether the corresponding operation is to be performed on all or a subset of the data

elements of the packed data operand(s). In this manner, the same opcodes may be used for both full-width packed data operations and partial-width packed data operations. Alternatively, one set of opcodes may be used for partial-width packed data operations and a different set of opcodes may be used for full-width packed data operations.

In any event, if the instruction is a conventional full-width packed data instruction, then at step 330, a packed data result is determined by performing the operation specified by the instruction on each of the data elements in the operand(s). However, if the instruction is a partial-width packed data instruction, then at step 340, a first portion of the result is determined by performing the operation specified by the instruction on a subset of the data elements and the remainder of the result is set to one or more predetermined values. In one embodiment, the predetermined value is the value of the corresponding data element in one of the operands. That is, data elements may be "passed through" from data elements of one of the operands to corresponding data elements in the packed data result. In another embodiment, the data elements in the remaining portion of the result are all cleared (zeroed). Exemplary logic for performing the passing through of data elements from one of the operands to the result and exemplary logic for clearing data elements in the result are described below.

Figure 4 conceptually illustrates the result of executing a partial-width packed data instruction according to various embodiments of the invention. In this example, an operation is performed on data elements of two logical

source registers 410 and 420 by an execution unit 440. The execution unit 440 includes circuitry and logic for performing the operation specified by the instruction. In addition, the execution unit 440 may include selection circuitry that allows the execution unit 440 to operate in a partial-width packed data mode or a full-width packed data mode. For instance, the execution unit 440 may include pass through circuitry to pass data elements from one of the logical source registers 410, 420 to the logical destination register 430, or clearing circuitry to clear one or more data elements of the logical destination register 430, etc. Various other techniques may also be employed to affect the result of the operation, including forcing one of the inputs to the operation to a predetermined value, such as a value that would cause the operation to perform its identity function or a value that may pass through arithmetic operations without signaling an exception (e.g., a quiet not-a-number (QNaN)).

In the example illustrated, only the result (Z_0) of the operation on the first pair of data elements (X_0 and Y_0) is stored in the logical destination register 430. Assuming the execution unit 440 includes pass through logic, the remaining data elements of the logical destination register 430 are set to values from corresponding data elements of logical source register 410 (i.e., X_3, X_2, and X_1). While the logical destination register 430 is shown as a separate logical register, it is important to note that it may concurrently serve as one of the logical source registers 410, 420. Therefore, it should be appreciated that setting data elements of the logical destination register 430 to

values from one of the logical source registers 410, 420 in this context may include doing nothing at all. For example, in the case that logical source register 410 is both a logical source and destination register, various embodiments may take advantage of this and simply not touch one or more of the data elements which are to be passed through.

Alternatively, the execution unit 440 may include clearing logic. Thus, rather than passing through values from one of the logical source registers to the logical destination register 430, those of the data elements in the result that are unnecessary are cleared. Again, in this example, only the result (Z_0) of the operation on the first pair of data elements (X_0 and Y_0) is stored in the logical destination register 430. The remaining data elements of the logical destination register 430 are "cleared" (e.g., set to zero, or any other predetermined value for that matter).

Full-Width Hardware

Figures 5A - 5C conceptually illustrate execution units 540, 560 and 580, respectively, which may execute both full-width packed data and partial-width packed data instructions. The selection logic included in the execution units of Figures 5A and 5C represent exemplary pass through logic, while the selection logic of Figure 5B is representative of clearing logic that may be employed. In the embodiments depicted, the execution units 540, 560, and 580 each include appropriate logic, circuitry and/or firmware for concurrently

performing an operation 570, 571, and 572 on the full-width of the operands (X and Y).

Referring now to **Figure 5A**, the execution unit 540 includes selection logic (e.g., multiplexers (MUXes) 555-557) for selecting between a value produced by the operation 570 and a value from a corresponding data element of one of the operands. The MUXes 555-557 may be controlled, for example, by a signal that indicates whether the operation currently being executed is a full-width packed data operation or a partial-width packed data operation. In alternative embodiments, additional flexibility may be achieved by including an additional MUX for data element 0 and/or independently controlling each MUX. Various means of providing MUX control are possible. According to one embodiment, such control may originate or be derived from the instruction itself or may be provided via immediate values. For example, a 4-bit immediate value associated with the instruction may be used to allow the MUXes 555-557 to be controlled directly by software. Those MUXes corresponding to a one in the immediate value may be directed to select the result of the operation while those corresponding to a zero may be caused to select the pass through data. Of course, more or less resolution may be achieved in various implementations by employing more or less bits to represent the immediate value.

Turning now to **Figure 5B**, the execution unit 540 includes selection logic (e.g., MUXes 565-567) for selecting between a value produced by an

operation 571 and a predetermined value (e.g., zero). As above, the MUXes 565-567 may be under common control or independently controlled.

The pass through logic of Figure 5C (e.g., MUXes 575-576) selects between a data element of one of the operands and an identity function value 590. The identity function value 590 is generally chosen such that the result of performing the operation 572 between the identity function value 590 and the data element is the value of the data element. For example, if the operation 572 was a multiply operation, then the identity function value 590 would be 1. Similarly, if the operation 572 was an add operation, the identity function value 590 would be 0. In this manner, the value of a data element can be selectively passed through to the logical destination register 430 by causing the corresponding MUX 575-577 to output the identity function value 590.

In the embodiments described above, the circuitry was hardwired such that the partial-width operation was performed on the least significant data element portion. It is appreciated that the operation may be performed on a different data element portions than illustrated. Also, as described above, the data elements to be operated upon may be made to be software configurable by coupling all of the operations to a MUX or the like, rather than simply a subset of the operations as depicted in Figures 5A-5C. Further, while pass through and clearing logic are described as two options for treating resulting data elements corresponding to operations that are to be disregarded, alternative embodiments may employ other techniques. For example, a

QNaN may be input as one of the operands to an operation whose result is to be disregarded. In this manner, arithmetic operations compliant with the IEEE 754 standard, IEEE std. 754-1985, published March 21, 1985, will propagate a NaN through to the result without triggering an arithmetic exception.

While no apparent speed up would be achieved in the embodiments described above since the full-width of the operands can be processed in parallel, it should be appreciated that power consumption can be reduced by shutting down those of the operations whose results will be disregarded. Thus, significant power savings may be achieved. Additionally, with the use of QNaNs and/or identity function values a predictable exception model may be maintained by preventing exceptions from being triggered by data elements that are not part of the partial-width packed data operation. Therefore, reported exceptions are limited to those raised in connection with the data element(s) upon which the partial-width packed data operation purports to operate.

Figure 6 illustrates a current processor implementation of an arithmetic logic unit (ALU) that can be used to execute full-width packed data instructions. The ALU of Figure 6 includes the circuitry necessary to perform operations on the full width of the operands (i.e., all of the data elements). Figure 6 also shows that the ALU may contain one or more different types of execution units. In this example, the ALU includes two different types of execution units for respectively performing different types of operations (e.g., certain ALUs use separate units for performing ADD and MUL operations).

The ADD execution unit and the MUL execution unit are respectively capable of operating as four separate ADD execution units and four separate MUL execution units. Alternatively, the ALU may contain one or more Multiply Accumulate (MAC) units, each capable of performing more than a single type of operation. While the following examples assume the use of ADD and MUL execution units and floating point operations, it is appreciated other execution units such as MAC and/or integer operations may also be used. Further, it may be preferable to employ a partial-width implementation (e.g., an implementation with less than a one-to-one correspondence between execution units and data elements) and additional logic to coordinate reuse of the execution units as described below.

Partial-Width Hardware and "Staggered Execution"

Figures 7A-7B conceptually illustrate a full-width packed data operation and a partial-width packed data operation being performed in a "staggered" manner, respectively. "Staggered execution" in the context of this embodiment refers to the process of dividing each of an instruction's operands into separate segments and sequentially processing each segment using the same hardware. The segments are sequentially processed by introducing a delay into the processing of the subsequent segments. As illustrated in Figures 7A-7B, in both cases, the packed data operands are divided into a "high order segment" (data elements 3 and 2) and a "low order segment" (data elements 1 and 0). In the example of Figure 7A, the low order

segment is processed while the high order segment is delayed. Subsequently, the high order segment is processed and the full-width result is obtained. In the example of Figure 7B, the low order segment is processed, while whether the high order data segment is processed depends on the implementation. For example, the high order data segment may not need to be processed if the corresponding result is to be zeroed. Additionally, it is appreciated that if the high order data segment is not processed, then both the high and low order data segments may be operated upon at the same time. Similarly, in a full-width implementation (e.g., an implementation with a one-to-one correspondence between execution units and data elements) the high and low order data segments may be processed concurrently or as shown in Figure 7A.

Additionally, although the following embodiments are described as having only ADD and MUL execution units, other types of execution units such as MAC units may also be used.

While there are a number of different ways in which the staggered execution of instructions can be achieved, the following sections describe two exemplary embodiments to illustrate this aspect of the invention. In particular, both of the described exemplary embodiments receive the same macro instructions specifying logical registers containing 128 bit operands.

In the first exemplary embodiment, each macro instruction specifying logical registers containing 128 bit operands causes the full-width of the operands to be accessed from the physical registers. Subsequent to accessing the full-width operands from the registers, the operands are divided into the

low and high order segments (e.g., using latches and multiplexers) and sequentially executed using the same hardware. The resulting half-width results are collected and simultaneously written to a single logical register.

In contrast, in the second exemplary embodiment each macro instruction specifying logical registers containing 128 bit operands is divided into at least two micro instructions that each operate on only half of the operands. Thus, the operands are divided into a high and low order segment and each micro instruction separately causes only half of the operands to be accessed from the registers. This type of a division is possible in a SIMD architecture because each of the operands is independent from the other. While implementations of the second embodiment can execute the micro instructions in any order (either an in order or an out of order execution model), the micro instructions respectively cause the operation specified by the macro instruction to be independently or separately performed on the low and high order segments of the operands. In addition, each micro instruction causes half of the resulting operand to be written into the single destination logical register specified by the macro instruction.

While embodiments are described in which 128 bit operands are divided into two segments, alternative embodiments could use larger or smaller operands and/or divide those operands into more than two segments. In addition, while two exemplary embodiments are described for performing staggered execution, alternative embodiments could use other techniques.

First Exemplary Embodiment Employing "Staggered Execution"

Figure 8A conceptually illustrates circuitry within a processor according to a first embodiment that accesses full width operands from the logical registers but that performs operations on half of the width of the operands at a time. This embodiment assumes that the processor execution engine is capable of processing one instruction per clock cycle. By way of example, assume the following sequence of instructions is executed: ADD X, Y; MUL A, B. At time T, 128-bits of X and 128-bits of Y are each retrieved from their respective physical registers via ports 1 and 2. The lower order data segments, namely the lower 64 bits, of both X and Y are passed into multiplexers 802 and 804 and then on to the execution units for processing. The higher order data segments, the higher 64 bits of X and Y are held in delay elements M1 and M2. At time T+1, the higher order data segments of X and Y are read from delay elements M1 and M2 and passed into multiplexers 802 and 804 and then on to the execution units for processing. In general, the delay mechanism of storing the higher order data segments in delay elements M1 and M2 allows N-bit (N=64 in this example) hardware to process 2N-bits of data. The low order results from the execution unit are then held in delay element M3 until the high order results are ready. The results of both processing steps are then written back to register file 800 via port 3. Recall that in the case of a partial-width packed data operation one or more data elements of the low or high order results may be forced to a predetermined value (e.g., zero, the value of a

corresponding data element in one of X or Y, etc.) rather than the output of the ADD or MUL operation.

Continuing with the present example, at time T+1, the MUL instruction may also have been started. Thus, at time T+1, 128-bits of A and B may each have been retrieved from their respective registers via ports 1 and 2. The lower order data segments, namely the lower 64-bits, of both A and B may be passed into multiplexers 806 and 808. After the higher order bits of X and Y are removed from delay elements M1 and M2 and passed into multiplexers 806 and 808, the higher order bits of A and B may be held in storage in delay elements M1 and M2. The results of both processing steps is written back to register file 800 via port 3.

Thus, according to an embodiment of the invention, execution units are provided that contain only half the hardware (e.g. two single precision ADD execution units and two single precision MUL execution units), instead of the execution units required to process the full width of the operands in parallel as found in a current processor. This embodiment takes advantage of statistical analysis showing that multimedia applications utilize approximately fifty percent ADD instructions and fifty percent MUL instructions. Based on these statistics, this embodiment assumes that multimedia instructions generally follow the following pattern: ADD, MUL, ADD, MUL, etc.. By utilizing the ADD and MUL execution units in the manner described above, the present embodiment provides for an optimized

use of the execution units, thus enabling comparable performance to the current processor, but at a lower cost.

Figure 8B is a timing chart that further illustrates the circuitry of Figure 8A. More specifically, as illustrated in Figure 8B, when instruction "ADD X, Y" is issued at time T, the two ADD execution units first perform ADDs on the lower order data segments or the lower two packed data elements of Figure 1, namely X_0Y_0 and X_1Y_1. At time T + 1, the ADD operation is performed on the remaining two data elements from the operands, by the same execution units, and the subsequent two data elements of the higher order data segment are added, namely X_2Y_2 and X_3Y_3. While the above embodiment is described with reference to ADD and MUL operations using two execution units, alternate embodiments may use any number of execution units and/or execute any number of different operations in a staggered manner.

According to this embodiment, 64-bit hardware may be used to process 128-bit data. A 128-bit register may be broken into four 32-bit elements, each of which represents a separate 32-bit value. At time T, the two ADD execution units perform ADDs first on the two lower 32-bit values, followed by an ADD on the higher 32-bit values at time T+1. In the case of a MUL operation, the MUL execution units behave in the same manner. This ability to use currently available 64-bit hardware to process 128-bit data represents a significant cost advantage to hardware manufacturers.

As described above, the ADD and MUL execution units according to the present embodiment are reused to reexecute a second ADD or MUL operation at a subsequent clock cycle. Of course, in the case of a partial-width packed data instruction, the execution units are reused but the operation is not necessarily reexecuted since power to the execution unit may be selectively shut down. At any rate, as described earlier, in order for this re-using or "staggered execution" to perform efficiently, this embodiment takes advantage of the statistical behavior of multimedia applications.

If a second ADD instruction follows a first ADD instruction, the second ADD may be delayed by a scheduling unit to allow the ADD execution units to complete the first ADD instruction, or more specifically on the higher order data segment of the first ADD instruction. The second ADD instruction may then begin executing. Alternatively, in an out-of-order processor, the scheduling unit may determine that a MUL instruction further down the instruction stream may be performed out-of-order. If so, the scheduling unit may inform the MUL execution units to begin processing the MUL instruction. If no MUL instructions are available for processing at time T+1, the scheduler will not issue an instruction following the first ADD instruction, thus allowing the ADD execution units time to complete the first ADD instruction before beginning the second ADD instruction.

Yet another embodiment of the invention allows for back-to-back ADD or MUL instructions to be issued by executing the instructions on the same execution units on half clock cycles instead of full clock cycles. Executing an

instruction on the half clock cycle effectively "double pumps" the hardware, i.e. makes the hardware twice as fast. In this manner, the ADD or MUL execution units may be available during each clock cycle to process a new instruction. Double pumped hardware would allow for the hardware units to execute twice as efficiently as single pumped hardware that executes only on the full clock cycle. Double pumped hardware requires significantly more hardware, however, to effectively process the instruction on the half clock cycle.

It will be appreciated that modifications and variations of the invention are covered by the above teachings and within the purview of the appended claims without departing from the spirit and intended scope of the invention. For example, although only two execution units are described above, any number of logic units may be provided.

Second Exemplary Embodiment Employing "Staggered Execution"

According to an alternate embodiment of the invention, the staggered execution of a full width operand is achieved by converting a full width macro instruction into at least two micro instructions that each operate on only half of the operands. As will be described further below, when the macro instruction specifies a partial-width packed data operation, better performance can be achieved by eliminating micro instructions that are not necessary for the determination of the partial-width result. In this manner, processor resource constraints are reduced and the processor is not unnecessarily occupied with inconsequential micro instructions. Although

the description below is written according to a particular register renaming method, it will be appreciated that other register renaming mechanisms may also be utilized consistent with the invention. The register renaming method as described below assumes the use of a Register Alias Table (RAT), a Reorder Buffer (ROB) and a retirement buffer, as described in detail in U.S. Patent No. 5,446,912. Alternate register renaming methods such as that described in U.S. Patent No. 5,197,132 may also be implemented.

Figure 9 conceptually illustrates one embodiment of a pipeline to perform operations on operands in a "staggered" manner by converting a macro instruction into a plurality of micro instructions that each processes a portion of the full width of the operands. It should be noted that various other stages of the pipeline, e.g. a prefetch stage, have not been shown in detail in order not to unnecessarily obscure the invention. As illustrated, at the decode stage of the pipeline, a full width macro instruction is received, specifying logical source registers, each storing a full width operand (e.g. 128-bit). By way of example, the described operands are 128-bit packed floating point data operands. In this example, the processor supports Y logical registers for storing packed floating point data. The macro instruction is converted into micro instructions, namely a "high order operation" and a "low order operation," that each cause the operation of the macro instruction to be performed on half the width of the operands (e.g., 64 bits).

The two half width micro instructions then move into a register renaming stage of the pipeline. The register renaming stage includes a

variety of register maps and reorder buffers. The logical source registers of each micro instruction are pointers to specific register entries in a register mapping table (e.g. a RAT). The entries in the register mapping table in turn point to the location of the physical source location in an ROB or in a retirement register. According to one embodiment, in order to accommodate the half width high and low order operations described above, a RAT for packed floating point data is provided with Y*2 entries. Thus, for example, instead of a RAT with the entries for 8 logical registers, a RAT is created with 16 entries, each addressed as "high" or "low." Each entry identifies a 64-bit source corresponding to either a high or a low part of the 128-bit logical register.

Each of the high and low order micro instructions thus has associated entries in the register mapping table corresponding to the respective operands. The micro instructions then move into a scheduling stage (for an out of order processor) or to an execution stage (for an in order processor). Each micro instruction retrieves and separately processes a 64-bit segment of the 128-bit operands. One of the operations (e.g. the lower order operation) is first executed by the 64-bit hardware units. Then, the same 64-bit hardware unit executes the higher order operation. It should be appreciated that zero or more instructions may be executed between the lower and higher order operations.

Although the above embodiment describes the macro instruction being divided into two micro instructions, alternate embodiments may divide the

macro instruction into more micro instruction. While Figure 9 shows that the packed floating point data is returned to a retirement register file with Y*2 64-bit registers, each designated as high or low, alternate embodiments may use a retirement register file with Y 128-bit registers. In addition, while one embodiment is described having a register renaming mechanism with a reorder buffer and retirement register files, alternate embodiments may use any register renaming mechanism. For example, the register renaming mechanism of U.S. Patent No. 5,197,132 uses a history queue and backup map.

Figure 10 is a timing chart that further illustrates the embodiment described in Figures 9. At time T, a macro instruction "ADD X, Y" enters the decode stage of the pipeline of Figure 9. By way of example, the macro instruction here is a 128-bit instruction. The 128-bit macro instruction is converted into two 64-bit micro instructions, namely the high order operation, "ADD X_H, Y_H" and the low order operation, "ADD X_L Y_L." Each micro instruction then processes a segment of data containing two data elements. For example, at time T, the low order operation may be executed by a 64-bit execution unit. Then at a different time (e.g., time T+N), the high order operation is executed by the same 64-bit execution unit. This embodiment of the invention is thus especially suitable for processing 128-bit instructions using existing 64-bit hardware systems without significant changes to the hardware. The existing systems are easily extended to include a new map to handle packed floating point, in addition to the existing logical register maps.

Referring now to Figure 11, decoding logic that may be employed according to one embodiment of the invention is described. Briefly, in the embodiment depicted, a plurality of decoders 1110, 1120, and 1130 each receive a macro instruction and convert it into a micro instruction. Then the micro operating are sent down the remainder of the pipeline. Of course, N micro instructions are not necessary for the execution of every macro instruction. Therefore, it is typically the case that only a subset of micro instructions are queued for processing by the remainder of the pipeline.

As described above, packed data operations may be implemented as two half width micro instructions (e.g., a high order operation and a low order operation). Rather than independently decoding the macro instruction by two decoders to produce the high and low order operations as would be typically required by prior processor implementations, as a feature of the present embodiment both micro instructions may be generated by the same decoder. In this example, this is accomplished by replication logic 1150 which replicates either the high or low order operation and subsequently modifies the resulting replicated operation appropriately to create the remaining operation. Importantly, as was described earlier, by carefully encoding the register address space, the registers referenced by the micro instructions (e.g., the logical source and destination registers) can be made to differ by a single bit. As a result, the modification logic 1160 in its most simple form may comprise one or more inverters to invert the appropriate bits to produce a high order operation from a low order operation and vice versa. In any

event, the replicated micro instruction is then passed to multiplexer 1170. The multiplexer 1170 also receives a micro instruction produced by decoder 1120. In this example, the multiplexer 1170, under the control of a validity decoder 1180, outputs the replicated micro instruction for packed data operations (including partial-width packed data operations) and outputs the micro instruction received from decoder 1120 for operations other than packed data operations. Therefore, it is advantageous to optimize the opcode map to simplify the detection of packed data operations by the replication logic 1150. For example, if only a small portion of the of the macro instruction needs to be examined to distinguish packed data operations from other operations, then less circuitry may be employed by the validity decoder 1180.

In an implementation that passes through source data elements to the logical destination register for purposes of executing partial-width packed data operations, in addition to selection logic similar to that described with respect to Figures 5A and 5C, logic may be included to eliminate ("kill") one of the high or low order operations. Preferably, for performance reasons, the extraneous micro instruction is eliminated early in the pipeline. This elimination may be accomplished according to the embodiment depicted by using a micro instruction selection signal output from micro instruction length determination circuitry 1190. The micro instruction length determination logic 1190 examines a portion of the macro instruction and produces the micro instruction selection signal which indicates a particular

combination of one or more micro instructions that are to proceed down the pipeline. In the case of a scalar SIMD instruction, only one of the resulting high and low order operations will be allowed to proceed. For example, the micro instruction selection signal may be represented as a bit mask that identifies those of the micro instructions that are to be retained and those that are to be eliminated. Alternatively, the micro instruction selection signal may simply indicate the number of micro instructions from a predetermined starting point that are to be eliminated or retained. Logic required to perform the elimination described above will vary depending upon the steering mechanism that guides the micro instructions through the remainder of the pipeline. For instance, if the micro instructions are queued, logic would may be added to manipulate the head and tail pointers of the micro instruction queue to cause invalid micro instructions to be overwritten by subsequently generated valid micro instructions. Numerous other elimination techniques will be apparent to those of ordinary skill in the art.

Although for simplicity only a single macro instruction is shown as being decoded at a time in the embodiment depicted, in alternative embodiments multiple macro instructions may be decoded concurrently. Also, it is appreciated that micro instruction replication has broader applicability than that illustrated by the above embodiment. For example, in a manner similar to that described above, full-width and partial-width packed data macro instructions may be decoded by the same decoder. If a prefix is used to distinguish full-width and partial width packed data macro

instructions, the decoder may simply ignore the prefix and decode both types of instructions in the same manner. Then, the appropriate bits in the resulting micro operations may be modified to selectively enable processing for either all or a subset of the data elements. In this manner, full-width packed data micro operations may be generated from partial-width packed data micro operations or vice versa, thereby reducing complexity of the decoder.

Thus, a method and apparatus for efficiently executing partial-width packed data instructions are disclosed. These specific arrangements and methods described herein are merely illustrative of the principles of the invention. Numerous modifications in form and detail may be made by those of ordinary skill in the art without departing from the scope of the invention. Although this invention has been shown in relation to a particular preferred embodiment, it should not be considered so limited. Rather, the invention is limited only by the scope of the appended claims.

CLAIMS

1. A processor comprising:

a plurality of registers;

a register renaming unit coupled to the plurality of registers to provide an architectural register file to store packed data operands, each of said packed data operands having a plurality of data elements;

a decoder, coupled to said register renaming unit, to decode a first and second set of instructions that each specify one or more registers in the architectural register file, each instruction in the first set of instructions specifying operations on all of the data elements stored in the specified one or more registers, each of the second set of instructions specifying an operation on only a subset of data element stored in a specified one or more registers; and

a partial-width execution unit, coupled to the decoder to execute operations specified by either of the first or the second set of instructions.

2. The processor of claim 1, wherein the subset of data elements stored in a specified one or more registers comprises corresponding least significant data elements.

3. The processor of claim 1, further comprising an execution unit to selectively perform a specified operation on one or more data elements

in the specified one or more registers depending upon which of the first or second set of instructions the specified operation is associated.

4. The procesor of claim 3, wherein the execution unit further comprises a plurality of multiplexers to select between a result of the specified operation and a predetermined value.

5. The processor of claim 3, wherein the execution unit further comprises a plurality of multiplexers to select between a data element of the one or more data elements and an identity function for input to the specified operation.

6. A method comprising the steps of:

receiving a single macro instruction specifying at least two logical registers in a packed data register file, wherein the two logical registers respectively store a first packed data operand and second packed data operand having corresponding data elements; and

independently operating on a first and second plurality of the corresponding data elements from said first and second packed data operands at different times using the same circuit to independently generate a first and second plurality of resulting data elements by

performing an operation specified by the single macro instruction on at least one pair of corresponding data elements in the first and second plurality corresponding

data elements to produce at least one resulting data element of the first and second plurality of resulting data elements, and

setting remaining resulting data elements of the first and second plurality of resulting data elements to one or more predetermined values; and

storing the first and second plurality of resulting data elements in a single logical register as a third packed data operand.

7. The method of claim 6, wherein the one or more predetermined values comprise values of data elements from either the first packed data operand or the second packed data operand.

8. The method of claim 6, wherein the one or more predetermined values comprise zero.

9. The method of claim 6, wherein the one or more predetermined values comprise a not-a-number (NaN) indication.

10. A processor substantially as herein described with reference to and as shown in each of the embodiments shown in Figures 2A-11 of the accompanying drawings.

11. A method substantially as herein described with reference to and as shown in each of the embodiments shown in Figures 2A-11 of the accompanying drawings.

The Patent Office

INVESTOR IN PEOPLE

Application No: GB 9907221.7
Claims searched: 1

Examiner: Leslie Middleton
Date of search: 28 October 1999

Patents Act 1977
Search Report under Section 17

Databases searched:

UK Patent Office collections, including GB, EP, WO & US patent specifications, in:

UK Cl (Ed.Q): G4A (AVL)

Int Cl (Ed.6): G06F 9/302

Other: Online: EPODOC, PAJ, WPI / EPOQUE

Documents considered to be relevant:

Category	Identity of document and relevant passage	Relevant to claims
A	WO 9722924 A1 (Intel Corpn.) See Fig. 3A, and column 9	
A	WO 97/22923 A1 (Intel Corpn.) See Fig. 3A, and cols.6,9.	
A	WO 97/22921 A1 (Intel Corpn.) See Fig. 3A, & pp.15,21.	

X	Document indicating lack of novelty or inventive step	A	Document indicating technological background and/or state of the art.
Y	Document indicating lack of inventive step if combined with one or more other documents of same category.	P	Document published on or after the declared priority date but before the filing date of this invention.
&	Member of the same patent family	E	Patent document published on or after, but with priority date earlier than, the filing date of this application.